Clase 6 — 09.12.25

#IntelliJIDEA #JAVA #XML #DOM #SAX

👨‍🏫 **Profesor:** Álvaro García Gutiérrez
📘 **Acceso a Datos**
🗓️ **Clase 6 — 09/12/2025**
🎯 **Tema: Repaso autoevaluación (Ficheros, Streams y RandomAccessFile)**

✅ Cuestionario

1️⃣ Si necesitas leer líneas de un archivo de texto, ¿qué clase te simplifica la vida?

Opciones:

- a) RandomAccessFile con `readLine()` (siempre recomendado)
- b) FileReader con `readLine()`
- c) Scanner.nextInt()
- d) BufferedReader con `readLine()`

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
Cuando lees texto "por líneas", lo que realmente estás pidiendo es: *"dame caracteres hasta detectar un separador de línea"*. Hacer eso manualmente con lecturas carácter a carácter es posible, pero es tedioso y propenso a errores (gestión de `\n`, `\r\n`, líneas vacías, última línea sin salto, etc.).

`BufferedReader` te da exactamente esa abstracción con `readLine()` y, además, incorpora un **búfer**. El búfer significa que la clase no va al sistema operativo por cada carácter; en lugar de eso, **lee un bloque grande** y lo va entregando poco a poco. En E/S, reducir llamadas al SO suele ser la diferencia entre "va bien" y "se arrastra" cuando el archivo crece.

- `FileReader` NO tiene `readLine()`. Solo tiene `read()` para caracteres. Por eso la opción b) es incorrecta por API.
- `Scanner` es útil para parsear tokens (enteros, palabras) y expresiones, pero suele ser más pesado: internamente usa patrones, conversión y validaciones que no necesitas si solo quieres líneas.
- `RandomAccessFile.readLine()` existe, pero no es la opción "natural" para lectura de texto moderna; está más ligado a acceso aleatorio y binario. Además, en muchos cursos se desaconseja usarlo para texto como "primera elección" y se prefiere `BufferedReader`.

✅ Patrón típico correcto:

```
try (BufferedReader br = new BufferedReader(new FileReader("datos.txt"))) {
    String linea;
    while ((linea = br.readLine()) != null) {
        // Procesas la línea completa (sin \n)
        System.out.println(linea);
```

```
    }
}
```

💥 Error típico en examen: usar `FileReader` y pretender `readLine()` (no compila) o usar `Scanner` sin entender el impacto en rendimiento.

2️⃣ ¿Por qué los registros de tamaño fijo permiten acceso directo al registro *i*?

Opciones:

- a) Porque `seek()` solo funciona con strings de tamaño fijo
- b) Porque todos los registros se almacenan como texto
- c) Porque `length()` devuelve el número de registros
- d) Porque el offset se calcula como `i * tamRegistro`

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
El concepto clave es **offset** (desplazamiento en bytes). En un fichero de registros fijos, cada registro ocupa exactamente `tamRegistro` bytes. Eso convierte el fichero en algo parecido a un array en disco: si todos ocupan lo mismo, el inicio del registro `i` es matemáticamente determinable.

- Registro 0 empieza en `0`
- Registro 1 empieza en `tamRegistro`
- Registro 2 empieza en `2*tamRegistro`
- …
- Registro i empieza en `i*tamRegistro`

Esto permite **acceso directo** (random access) sin leer lo anterior. `seek(offset)` mueve el puntero a esa posición y luego lees el registro completo en el orden correcto.

✅ Ejemplo:

```
int tamRegistro = 36;
int i = 10;
long offset = (long) i * tamRegistro; // long para evitar overflow
raf.seek(offset);                     // salto directo
```

⚠️ Importante: el acceso directo solo funciona si:

1. el tamaño del registro es fijo **de verdad** (mismos bytes siempre)
2. lees y escribes el registro en **el mismo formato** (orden/tipos)

💥 Error típico: usar `writeUTF()` dentro de un registro “supuestamente fijo”. Eso lo rompe porque es de tamaño variable.

3️⃣ ¿Qué excepción es típica al pasarse del final con `readInt()` / `readDouble()`?

Opciones:

- a) NullPointerException

- b) ArithmeticException
- c) EOFException
- d) IllegalArgumentException

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
`readInt()` necesita leer **4 bytes**. `readDouble()` necesita **8 bytes**. Si el puntero está en una posición donde no quedan suficientes bytes para completar esa lectura, Java lanza `EOFException` (End Of File).

No es un "error de cálculo", es un mecanismo estándar: el fichero no contiene más datos para ese tipo primitivo. Por eso es una excepción *típica* en lectura binaria si no controlas el final.

✅ Forma segura 1: controlar con puntero y tamaño:

```
while (raf.getFilePointer() + 4 <= raf.length()) { // +4 para int
    int n = raf.readInt();
}
```

✅ Forma común 2: bucle infinito y captura:

```
try {
    while (true) System.out.println(raf.readInt());
} catch (EOFException e) {
    // fin alcanzado
}
```

💥 Error típico: creer que `readInt()` devuelve algo "nulo" al final como en texto. En binario no hay `null`; hay excepción.

4️⃣ Para crear eficientemente un archivo de texto grande, ¿qué combinación es más adecuada?

Opciones:

- a) Scanner con System.out
- b) RandomAccessFile en modo `"r"`
- c) BufferedWriter sobre FileWriter
- d) FileWriter sin búfer

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Escribir texto grande implica muchas operaciones de E/S. Si escribes con `FileWriter` sin búfer, cada `write()` puede provocar una escritura pequeña al sistema operativo. Eso es ineficiente porque las llamadas de E/S son caras.

`BufferedWriter` amortigua esto: acumula caracteres en memoria y los vuelca en bloques. El resultado es:

- Menos llamadas al sistema (mucho más rápido)
- Escrituras más grandes y eficientes

Además, el patrón `BufferedWriter(FileWriter)` es el "clásico" para texto.

✅ Ejemplo:

```
try (BufferedWriter bw = new BufferedWriter(new FileWriter("grande.txt"))) {
    for (int i = 0; i < 1_000_00; i++) {
        bw.write("Linea " + i);
        bw.newLine();
    }
}
```

💥 Error típico: confundir “funciona” con “eficiente”. FileWriter funciona, pero penaliza mucho con archivos grandes.

5️⃣ ¿Qué hace `BufferedWriter.newLine()`?

Opciones:

- a) Inserta un salto de línea adecuado al sistema operativo
- b) Cierra el fichero
- c) Escribe siempre `\n`
- d) Vacía el búfer

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
En texto, el salto de línea no es universal. Hay sistemas que usan:

- Linux/macOS: `\n`
- Windows: `\r\n`

`newLine()` escribe el separador correcto del sistema, haciendo tu fichero más portable y evitando inconsistencias (por ejemplo, que un editor de Windows muestre el archivo “en una sola línea” si solo hay `\n` en un contexto concreto).

✅ Ejemplo:

```
bw.write("primera");
bw.newLine();      // portable
bw.write("segunda");
```

💥 Error típico: pensar que `newLine()` hace `flush()`. No. Solo escribe el separador. Para asegurar volcado inmediato: `flush()`.

6️⃣ ¿Cuál es la afirmación correcta sobre ficheros binarios frente a texto?

Opciones:

- a) Siempre usan saltos de línea
- b) Son más legibles por humanos
- c) Guardan los datos tal como están en memoria y son más compactos
- d) Requieren separadores como `;`

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
En un fichero binario, un `int` se guarda como 4 bytes (su representación binaria). En texto, ese mismo `int` se guarda como caracteres ('1', '2', '3'…), lo cual:

- ocupa más
- requiere parseo al leer (convertir texto → número)
- es más lento y más propenso a errores de formato

En binario:

- el tamaño es fijo por tipo
- la lectura/escritura es directa
- es más compacto y eficiente

En texto:

- es legible
- es editable
- depende de codificación y formato

✅ Comparación:

```
raf.writeInt(123);        // 4 bytes exactos
bw.write("123");          // 3 chars (+ posible codificación)
```

💥 Error típico: querer "ver" el binario con un editor de texto y pensar que está corrupto. No: es binario.

7️⃣ Tras escribir un `int` y luego un `double`, ¿qué offsets esperas?

Opciones:

- a) 2 y 6
- b) 4 y 12
- c) 1 y 9
- d) 8 y 16

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
El puntero avanza exactamente los bytes que escribes:

- `writeInt()` escribe 4 bytes → puntero pasa a 4
- `writeDouble()` escribe 8 bytes → puntero pasa a 12

Los offsets "esperados" normalmente se interpretan como "posición acumulada tras cada escritura".

✅ Ejemplo:

```
raf.seek(0);
raf.writeInt(10);      // ahora puntero = 4
System.out.println(raf.getFilePointer());

raf.writeDouble(2.5);  // ahora puntero = 12
System.out.println(raf.getFilePointer());
```

💥 Error típico: pensar en “número de elementos” en vez de bytes. `RandomAccessFile` trabaja con bytes.

8️⃣ Para asegurar 10 caracteres exactos en un nombre, marca la correcta

Opciones:

- a) Ninguna recorta
- b) Ambas escriben UTF-8
- c) `setLength(10)` rellena con `\u0000`; `String.format` con espacios
- d) Ambas rellenan con espacios

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Aquí la clave es **cómo fuerzas longitud fija** y con qué carácter rellenas.

`StringBuffer#setLength(10)`

- Si la cadena es más corta, se rellena con `\u0000` (carácter nulo).
- Si es más larga, se recorta.
  Esto es útil para “reservar hueco” fijo, pero introduce `\u0000`, que luego puede aparecer al recuperar.

```
StringBuffer sb = new StringBuffer("Ana");
sb.setLength(10); // Ana\u0000\u0000\u0000...
```

`String.format("%-10s", "Ana")`

- Rellena con espacios a la derecha.
- Es más “amigable” si lo vas a imprimir y luego limpiar con `trim()`.

```
String fijo = String.format("%-10s", "Ana"); // "Ana       "
```

📌 En ficheros de registros fijos, cualquiera puede servir, pero debes saber:

- si rellenas con `\u0000`, al convertir a String quizá necesites limpiar nulos
- si rellenas con espacios, `trim()` suele bastar

💥 Error típico: creer que ambos rellenan igual. No: **nulos vs espacios**.

9️⃣ Si rellenas un nombre fijo con `StringBuffer#setLength(10)`, ¿qué carácter se añade?

Opciones:

- a) Espacio
- b) No rellena
- c) Tabulación
- d) `\u0000`

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
`StringBuffer` cuando aumenta longitud, rellena con el carácter nulo Unicode `\u0000`. Este carácter:

- no “se ve” al imprimir normalmente
- sí ocupa espacio (2 bytes si lo escribes como `char`)
- puede ensuciar comparaciones o visualización si no lo limpias

✅ Mini-demo conceptual:

```
StringBuffer sb = new StringBuffer("Pepe");
sb.setLength(6);
System.out.println(sb.length()); // 6
// El contenido real incluye 2 nulos al final
```

🔟 Para truncar un archivo binario a 8 bytes con RandomAccessFile…

Opciones:

- a) `setLength(8)`
- b) `close(8)`
- c) `length(8)`
- d) `seek(8)`

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
Truncar significa cambiar el **tamaño físico** del fichero. Solo `setLength(n)` lo hace.

- `seek(8)` mueve el puntero a 8, pero el fichero sigue midiendo lo mismo.
- `length()` devuelve el tamaño; no lo cambia.
- `close(8)` no existe.

✅ Ejemplo:

```
RandomAccessFile raf = new RandomAccessFile("datos.bin", "rw");
raf.setLength(8); // ahora el fichero mide exactamente 8 bytes
```

📌 Implicación práctica: si antes medía 100 bytes, los bytes del 8 al 99 desaparecen. Si luego intentas leerlos, no existen.

1️⃣1️⃣ ¿Por qué se declara `int c` y no `char c` al usar `read()`?

Opciones:

- a) Ahorrar memoria
- b) Para imprimir en hexadecimal
- c) Porque `char` no existe
- d) Porque `read()` devuelve `-1` en EOF

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
`read()` devuelve un entero con dos significados posibles:

- 0..255: byte/caracter leído
- `-1`: fin de fichero (EOF)

Necesitas un tipo que pueda almacenar -1. `char` es unsigned (0..65535), así que no puede representar -1. Por eso se usa `int`.

✅ Patrón estándar:

```
int c;
while ((c = fr.read()) != -1) {
    char ch = (char) c;
    System.out.print(ch);
}
```

💥 Error típico: usar `char` y entrar en bucles infinitos o lecturas incorrectas.

1️⃣2️⃣ Número de registros en un fichero de registros fijos

Opciones:

- a) `length() / tamRegistro`
- b) `getFilePointer() / tamRegistro`
- c) `tamRegistro / length()`
- d) `seek() / tamRegistro`

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`length()` es el tamaño total del fichero en bytes. Si cada registro ocupa `tamRegistro` bytes y están almacenados uno detrás de otro, el número de registros completos es:

`numRegistros = length / tamRegistro`

Es la misma idea que “tamaño total / tamaño de cada bloque”.

```
long num = raf.length() / tamRegistro;
```

⚠️ `getFilePointer()` no vale para contar registros totales porque depende de dónde estés leyendo, no del tamaño real del fichero.

1️⃣3️⃣ Evitar overflow al calcular el offset

Opciones:

- a) Multiplicar en `long`
- b) Usar siempre `int`
- c) Dividir antes
- d) Sumar en un bucle

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
Offsets son bytes. En ficheros grandes, `i * tamRegistro` puede superar 2.147.483.647 (límite de `int`). Incluso aunque el resultado final quepa en `long`, si multiplicas en `int` primero, ya se rompió.

La solución: forzar la operación a `long` desde el inicio.

```
long offset = (long) i * tamRegistro; // fuerza multiplicación en 64-bit
raf.seek(offset);
```

💥 Error típico: calcular offset en `int` y que `seek()` vaya a posiciones incorrectas "sin avisar".

1️⃣4️⃣ Tamaño en bytes de `int`, `double`, `char`

Opciones:

- a) 4, 4, 2
- b) 4, 8, 2
- c) 2, 8, 1
- d) 8, 8, 1

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
En Java los tamaños de tipos primitivos están definidos por el lenguaje (no por el SO):

- `int` = 32 bits = 4 bytes
- `double` = 64 bits = 8 bytes
- `char` = 16 bits = 2 bytes (UTF-16 code unit)

Este punto es crucial para calcular registros fijos.

1️⃣5️⃣ Modo de RandomAccessFile que crea el archivo y permite leer/escribir

Opciones:

- a) append
- b) w
- c) rw
- d) r

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
RandomAccessFile solo admite modos `"r"` y `"rw"`:

- `"r"`: lectura únicamente
- `"rw"`: lectura y escritura. Si el fichero no existe, lo crea.

Además, solo en `"rw"` puedes hacer operaciones que alteren el tamaño (`setLength`) o escriban datos.

```
RandomAccessFile raf = new RandomAccessFile("datos.bin", "rw");
raf.writeInt(123);
```

1️⃣6️⃣ Ventaja de BufferedReader frente a FileReader

Opciones:

- a) Menos llamadas al sistema + `readLine()`
- b) Cambia codificación
- c) Convierte binario a texto
- d) Evita excepciones

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`FileReader` lee caracteres, pero sin búfer y sin utilidades de línea.
`BufferedReader` añade:

- búfer (rendimiento)
- `readLine()` (comodidad y claridad)
- `mark()` / `reset()` (en algunos casos)

Eso es exactamente lo que se suele pedir en ejercicios de lectura de texto.

1️⃣7️⃣ Abrir RandomAccessFile en modo `"r"`

Opciones:

- a) Solo escritura
- b) Solo lectura
- c) Lectura y escritura
- d) Ninguna

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
En modo `"r"`:

- puedes `readInt`, `readDouble`, etc.
- NO puedes `write...`
- NO puedes `setLength()`
- NO puedes alterar el fichero

Esto es importante porque muchos errores de clase vienen de intentar truncar o escribir en `"r"`.

1️⃣8️⃣ Cabecera con dos `int` (versión + contador)

Opciones:

- a) 16 bytes
- b) 4 bytes
- c) 8 bytes
- d) Depende del SO

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Una cabecera binaria típica puede guardar metadatos:

- versión del formato
- número de registros
- flags, etc.

Si la cabecera tiene 2 ints:

- 1 int = 4 bytes
- 2 ints = 8 bytes

✅ Ejemplo de escritura/lectura:

```
raf.seek(0);
raf.writeInt(1);    // version
raf.writeInt(100);  // contador

raf.seek(0);
int version = raf.readInt();
int count   = raf.readInt();
```

1️⃣9️⃣ Sobre `setLength(n)`

Opciones:

- a) Solo modo `"r"`
- b) Trunca o extiende (bytes nuevos a 0)
- c) Borra siempre
- d) Cambia codificación

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
`setLength(n)` es “control directo del tamaño físico del fichero”.

- Si `n` < tamaño actual → recorta, se pierde información del final.
- Si `n` > tamaño actual → amplía, y los bytes nuevos se rellenan con 0.

Esto se usa para:

- truncar (borrar contenido sobrante)
- “inicializar” o reservar espacio
- resetear fichero a vacío (`setLength(0)`)

```
raf.setLength(0);   // borra el contenido del fichero (lo deja en 0 bytes)
raf.setLength(100); // extiende; bytes nuevos a 0
```

2️⃣0️⃣ Diferencia entre `writeUTF` y `writeChars`

Opciones:

- a) Alineación
- b) Ninguna
- c) UTF con longitud vs chars fijos
- d) ASCII vs UTF

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Este es un punto clásico de “teoría con trampa” porque afecta directamente a registros fijos y offsets.

✅ `writeUTF(String)`

- Es **tamaño variable**.
- Escribe primero una longitud (en bytes) y luego los bytes del texto.
- El tamaño total depende del contenido.
  Resultado: **no puedes saltar al registro i con i*tamRegistro** si dentro hay UTFs.

```
raf.writeUTF("Ana");     // ocupa: 2 bytes de longitud + bytes del string (variable)
raf.writeUTF("Alejandro"); // ocupa más
```

✅ `writeChars(String)`

- Es **tamaño fijo por carácter**: cada `char` son 2 bytes.
- No escribe longitud.
- Si quieres 10 caracteres fijos, debes asegurarte tú (relleno/recorte).
  Resultado: perfecto para registros fijos.

```
String fijo10 = String.format("%-10s", "Ana"); // 10 chars
raf.writeChars(fijo10); // 10 * 2 = 20 bytes exactos
```

📌 Conclusión de examen:

- `writeUTF` → flexible pero rompe offsets fijos
- `writeChars` → rígido pero ideal para acceso directo

💥 Error típico: mezclar `writeUTF` al escribir y `readChars`/lecturas fijas al leer → puntero descuadrado.

2️⃣1️⃣ Método que devuelve la posición del puntero

Opciones:

- a) position()
- b) getFilePointer()
- c) length()
- d) available()

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
`RandomAccessFile` mantiene internamente un **puntero** que indica en qué byte exacto del fichero se realizará la siguiente lectura o escritura.
`getFilePointer()` devuelve ese valor en **bytes**, no en registros ni en caracteres.

Es fundamental distinguir:

- `getFilePointer()` → *dónde estoy ahora*
- `length()` → *cuánto mide el fichero*

```
raf.writeInt(10);
System.out.println(raf.getFilePointer()); // 4
```

💥 Error típico: usar `length()` pensando que devuelve la posición actual del puntero.

2️⃣2️⃣ Patrón `while ((c = fr.read()) != -1)`

Opciones:

- a) No mueve puntero
- b) Lee línea
- c) Resetea
- d) Lee carácter y avanza

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
Este patrón es la base de la lectura secuencial en Java. Cada llamada a `read()`:

- devuelve un carácter (o byte) como `int`
- **avanza automáticamente el puntero**

Cuando se llega al final del fichero, `read()` devuelve `-1`, permitiendo salir del bucle sin excepción.

```
int c;
while ((c = fr.read()) != -1) {
    System.out.print((char) c);
}
```

📌 Aquí no hay saltos ni reposicionamientos: es lectura **lineal** pura.

2️⃣3️⃣ ¿Por qué no puedes saltar “a ojo” con `writeUTF`?

Opciones:

- a) seek no funciona
- b) ASCII
- c) Tamaño variable
- d) Siempre 4 bytes

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
`writeUTF()` escribe:

1. una longitud
2. los bytes del string codificados

La longitud depende del contenido del texto, no del número de caracteres “a simple vista”. Esto rompe cualquier cálculo fijo del tipo `offset = i * tamaño`.

```
raf.writeUTF("Ana");        // ocupa menos
raf.writeUTF("Alejandro"); // ocupa más
```

📌 Conclusión clara:

- `writeUTF` → no compatible con acceso aleatorio por registros fijos
- `writeChars` → sí compatible

💥 Error típico: usar `writeUTF` y luego intentar `seek(i * tamRegistro)`.

2️⃣4️⃣ FileReader vs BufferedReader (eficiencia)

Opciones:

- a) Falso
- b) Verdadero

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
`BufferedReader` es más eficiente porque:

- reduce llamadas al sistema operativo
- lee bloques grandes y los entrega poco a poco
- evita overhead de E/S repetitiva

En archivos pequeños no se nota, pero en archivos medianos/grandes **sí**, y mucho.

📌 Por eso en ejercicios y en examen:

> FileReader + BufferedReader
> es el patrón correcto.

2️⃣5️⃣ Fichero de 3600 bytes con registros de 36 bytes

Opciones:

- a) 60
- b) 36
- c) 120
- d) 100

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
Aquí se aplica directamente la regla de registros fijos:

```
nº registros = tamaño fichero / tamaño registro
```

3600 / 36 = 100

📌 No hay resto → exactamente 100 registros completos.

💥 Error típico: confundir tamaño del registro con número de registros.

2️⃣6️⃣ Offset del registro *n* (36 bytes)

Opciones:

- a) n / 36
- b) n + 36
- c) 36 - n
- d) n * 36

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
El offset siempre se calcula como:

```
offset = índice * tamañoRegistro
```

Si el registro ocupa 36 bytes:

- registro 0 → offset 0
- registro 1 → offset 36
- registro n → offset n * 36

```
raf.seek((long) n * 36);
```

📌 El índice se asume **0-based**, como en arrays.

2️⃣7️⃣ Leer campos en distinto orden en binario

Opciones:

- a) Solo afecta a doubles
- b) Se pospone EOF
- c) No pasa nada
- d) Se descuadra el puntero

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
En binario **no hay separadores** ni marcas de campo. El lector debe saber exactamente:

- qué tipo viene ahora
- cuántos bytes ocupa

Si escribes:

```
raf.writeInt(10);
raf.writeDouble(2.5);
```

y luego lees:

```
double x = raf.readDouble(); // ERROR conceptual
```

El lector consume bytes incorrectos y el puntero queda desalineado para siempre.

📌 Regla de oro:

> El orden de lectura debe coincidir EXACTAMENTE con el de escritura.

2️⃣8️⃣ Uso de `getFilePointer() < length()`

Opciones:

- a) Devuelve registros
- b) Resetea
- c) Evita excepciones
- d) Evita pasar el final

✅ **Respuesta correcta:** d

📌 **Justificación MUY ampliada:**
Este patrón permite comprobar si **aún quedan bytes por leer** antes de intentar una lectura que podría lanzar `EOFException`.

```
while (raf.getFilePointer() < raf.length()) {
    int n = raf.readInt();
}
```

📌 Es una alternativa limpia a capturar excepciones como control de flujo.

2️⃣9️⃣ ¿Qué devuelve `length()`?

Opciones:

- a) Tamaño en bytes
- b) Registros
- c) Puntero
- d) Último índice

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`length()` devuelve el tamaño **físico real** del fichero en bytes, no depende del puntero ni del modo de apertura.

```
System.out.println(raf.length()); // bytes totales
```

📌 Es la base para:

- contar registros
- controlar EOF
- validar offsets

3️⃣0️⃣ ¿Qué hace `writeChars(String)`?

Opciones:

- a) Little-endian
- b) ASCII
- c) Cada char en 2 bytes
- d) UTF-8 con longitud

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
`writeChars()` escribe cada carácter Unicode (`char`) directamente como 2 bytes (UTF-16).
No guarda longitud ni separadores.

Esto hace que:

- el tamaño sea **predecible**
- sea ideal para registros fijos

```
raf.writeChars("AB"); // 2 chars → 4 bytes
```

📌 El lector debe saber **cuántos chars leer**.

3️⃣1️⃣ Leer más allá del final con `readInt()`

Opciones:

- a) EOFException
- b) IllegalStateException
- c) FileNotFoundException
- d) NumberFormatException

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`readInt()` necesita 4 bytes. Si el puntero está demasiado cerca del final y no hay suficientes bytes, Java lanza `EOFException` .

Esto es **comportamiento normal**, no un bug.

3️⃣2️⃣ Saltar al registro 10 de 48 bytes

Opciones:

- a) seek(10 + 48)
- b) seek(10L * 48)
- c) seek(10 / 48)
- d) seek(48 / 10)

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
Se aplica la fórmula estándar:

```
offset = índice * tamañoRegistro
```

Y se usa `long` para evitar overflow:

```
raf.seek(10L * 48);
```

3️⃣3️⃣ seek(4) y readInt() tras escribir 100, 200, 300

Opciones:

- a) 100
- b) Depende del SO
- c) 200
- d) 300

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Cada `int` ocupa 4 bytes:

Valor	Bytes
100	0–3
200	4–7
300	8–11

```
raf.seek(4);
int x = raf.readInt(); // 200
```

📌 El SO no influye en el tamaño de tipos primitivos Java.

3️⃣4️⃣ Tamaño del registro: int + char[10] + int + double

Opciones:

- a) 40
- b) 28
- c) 36
- d) 48

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**

Campo	Bytes
int	4
char[10]	20
int	4
double	8
Total	**36**

📌 Este cálculo es típico de examen y base para offsets.

3️⃣5️⃣ `setLength(100)` y luego `setLength(20)`

Opciones:

- a) 20 y 100
- b) 100 y 20; bytes nuevos a 0
- c) Siempre 0

- d) No cambia

✅ **Respuesta correcta:** b

📌 **Justificación MUY ampliada:**
Primero:

- el fichero se extiende a 100 bytes
- los nuevos bytes se rellenan con 0

Después:

- se trunca a 20 bytes
- se pierde todo lo que había más allá

3️⃣6️⃣ `setLength(0)` y el puntero

Opciones:

- a) Se sitúa en 0
- b) Lanza excepción
- c) Permanece
- d) Va al final original

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
El fichero queda vacío. El único puntero válido es 0.

📌 Es la forma “binaria” de borrar un fichero sin eliminarlo.

3️⃣7️⃣ Convertir `char[]` fijo a `String` y hacer `trim()`

Opciones:

- a) Añade salto
- b) Recupera longitud
- c) Elimina espacios/nulos
- d) Cambia codificación

✅ **Respuesta correcta:** c

📌 **Justificación MUY ampliada:**
Al leer chars fijos, suelen venir rellenos:

- espacios
- `\u0000`

Convertir a `String` y aplicar `trim()` limpia el resultado para uso normal.

```
String nombre = new String(chars).trim();
```

3️⃣8️⃣ `getFilePointer()` y `seek(p)`

Opciones:

- a) Mueve el puntero
- b) Trunca
- c) Escribe
- d) Devuelve tamaño

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`seek(p)` repositiona el puntero al byte `p`.
No escribe ni borra nada.

3️⃣9️⃣ Offset del cuarto entero (índice 3)

Opciones:

- a) 12
- b) 4
- c) 16
- d) 8

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
Índice 3 (0-based):

```
offset = 3 * 4 = 12
```

4️⃣0️⃣ Leer `char[10]` con `readUTF()` por error

Opciones:

- a) Se desalineará la lectura
- b) Se rellena con espacios
- c) Detecta límite
- d) Se corrige solo

✅ **Respuesta correcta:** a

📌 **Justificación MUY ampliada:**
`readUTF()` espera:

- primero una longitud
- luego bytes UTF variables

Si el fichero tiene chars fijos, `readUTF()` leerá bytes incorrectos, el puntero quedará mal posicionado y **todas las lecturas siguientes fallarán**.

📌 Este es uno de los errores más graves en binario.

✅ **Fin del cuestionario — 40/40 justificadas en profundidad**